

# Meta-Evaluation, Lazy Evaluation

## MIT 6.001 Recitation

What does it mean to say that `m-eval` and `m-apply` are "mutually-recursive procedures"?

### Letrec

---

Recall that `letrec` allows you to define recursive procedures in a `let`:

```
(letrec ((fact (lambda(n)
  (if (< n 2) 1
      (* n (fact (- n 1)))))))
  (fact 3))
```

By the way, why wouldn't `let` work in this case?

Write a syntactic transform implementation of `letrec`

### Scoping

---

Describe the behavior of the following code for the lexical vs. dynamic scoping implementations of the evaluator.

```
(define PRECISION 0.1)
(define close-enuf? (lambda (guess x)
  (< (abs (- (square guess) x)) PRECISION)))
(define improve (lambda (guess x) (/ (+ guess (/ x guess)) 2)))
(define sqrt-loop (lambda G X)
  (if (close-enuf? G X) G
      (sqrt-loop (improve G X) X)))
(define sqrt (lambda (x) (sqrt-loop 1.0 x)))

(define (f x)
  (define PRECISION 0.001)
  (sqrt x))
```

How could we achieve the same behavior using lexical scoping?

### Normal vs. applicative order

---

What does the following expression evaluate to using applicative order and normal order?

```
(letrec ((f (lambda (x) (cons x (f (+ 1 x))))))
  (car (f 1)))
```

Exhibit a program that you would expect to run much more slowly without memoization than with memoization.

### Type checking

---

How would you add the ability to check types for defensive evaluation?

You can do it to various degrees. You can even extend it to arbitrary predicates for the arguments.

```
;;; The 6.001 Meta-Circular Evaluator, Lazy Version
;;; The Core Evaluator
```

```
(define (l-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp) (lambda-body exp) env))
        ((begin? exp) (eval-sequence (begin-actions exp) env))
        ((application? exp)
         (l-apply (actual-value (operator exp) env) (operands exp) env))
        (else (error "Unknown expression type -- L-EVAL" exp))))

(define (l-apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure (list-of-arg-values arguments env)))
        ((compound-procedure? procedure)
         (eval-sequence (procedure-body procedure)
                          (extend-environment (procedure-parameters procedure)
                                              (list-of-delayed-args arguments env)
                                              (procedure-environment procedure))))
        (else (error "Unknown procedure type -- L-APPLY" procedure))))

(define (actual-value exp env) (force-it (l-eval exp env)))

(define (list-of-arg-values exps env)
  (if (no-operands? exps) '()
      (cons (actual-value (first-operand exps) env) (list-of-arg-values (rest-operands exps) env))))

(define (list-of-delayed-args exps env)
  (if (no-operands? exps) '()
      (cons (delay-it (first-operand exps) env) (list-of-delayed-args (rest-operands exps) env))))

(define (eval-if exp env)
  (if (actual-value (if-predicate exp) env)
      (l-eval (if-consequent exp) env)
      (l-eval (if-alternative exp) env)))

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (l-eval (first-exp exps) env))
        (else (l-eval (first-exp exps) env) (eval-sequence (rest-exps exps) env))))

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp) (l-eval (assignment-value exp) env) env))

(define (eval-definition exp env)
  (define-variable! (definition-variable exp) (l-eval (definition-value exp) env) env))

;;; Representing Expressions, procedures, environments: removed to save trees

;;; Representing Thunks

(define (delay-it exp env) (list 'thunk exp env))

(define (thunk? obj) (tagged-list? obj 'thunk))
(define (thunk-exp thunk) (cadr thunk))
(define (thunk-env thunk) (caddr thunk))

(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (actual-value (thunk-exp obj) (thunk-env obj))))
           (set-car! obj 'evaluated-thunk) (set-car! (cdr obj) result) (set-cdr! (cdr obj) '())
           result))
        ((evaluated-thunk? obj) (thunk-value obj))
        (else obj)))

(define (actual-value exp env) (force-it (l-eval exp env)))
(define (evaluated-thunk? obj) (tagged-list? obj 'evaluated-thunk))
(define (thunk-value evaluated-thunk) (cadr evaluated-thunk))
```